A Review of the OpenQASM Quantum Assembly Language

Karan Gurazada[1]

[1] *Stanford Online High School, Redwood City, CA*

Abstract: Quantum mechanics has traditionally relied on complex-valued probability amplitudes. In this paper, I examine why real-valued probability amplitudes are not sufficient to describe reality, and why a quantum mechanical theory based on quaternion- or octonion-valued probability amplitudes would be unphysical.

I. INTRODUCTION

OpenQASM stands for "open quantum assembly language." It is a quantum programming language originally developed by researchers at IBM[1], with contributions to the new version 3 from researchers at Amazon Web Services and the University of Sussex[2]. All versions of the language are imperative and statically typed and support both classical instructions, such as classical bit operators and `if` statements, and quantum instructions, such as qubit initialization, qubit measurement, and single- and multi-qubit unitary gates. Its syntax is heavily inspired by that of C. Programs written in OpenQASM 2.0 may be run for free on a real quantum computer on IBM's "quantum experience" web platform. OpenQASM 3.0 significantly broadens the language's scope and abilities, but at the present time there is no easily accessible way to execute programs written in it.

II. OPENQASM 2.0 SYNTAX

All files written in OpenQASM 2.0 should begin with the versionheader `OpenQASM 2.0;`. Most instructions end with semicolons, with `if` statements being the exception. To incorporate external libraries, use the `include` keyword; for example, `include "qelib1.inc";` includes IBM's standard quantum operations library. Comments in OpenQASM use similar syntax to C:

```
// this is a single line comment
/* and this
is a
multiline comment */
```

OpenQASM 2.0 has only four variable types: individual qubits and classical bits are represented by the `qubit` and `bit` types respectively, while n-bit quantum and classical registers are represented by the `qreg[n]` and `creg[n]` types respectively. To define a variable in OpenQASM, use the following syntax:

```
qubit quantum_bit; qreg quantum_register[4];
bit classical_bit; creg classical_register[4];
```

Both classical and quantum bits are initialized to 0 when defined. One may reset a quantum bit or register to zero using the `reset` keyword. If not using any external libraries, OpenQASM 2.0 only supports two types of quantum gates, a general unitary rotation and the CNOT gate. The unitary rotation is written

```
U(y, z, x) quantum_bit_or_register;
```

and represents a rotation of y radians around the y-axis, z radians around the z-axis, and x radians around the x-axis of a Bloch diagram. OpenQASM 2.0 does not have a "number" type for variables, but supports the built-in constant `pi`, base ten numbers, scientific notation, basic arithmetic operations (`+`, `-`, `*`, `/`), logarithms, trigonometric functions, exponents, and square roots. The CNOT gate is written

```
CX ctrl_bit_or_register, target_bit_or_register;
```

If using registers, the control and target registers must be of the same size. When acting on registers, OpenQASM gates perform the same action on every qubit in the register. OpenQASM is case-sensitive and ignores whitespace. Qubits can be measured using the `measure` keyword:

```
measure q_bit_or_register -> c_bit_or_register;
```

The classical bit or register that will hold the outcome of the measurement must be declared beforehand and must be the same size as the quantum bit or register being measured. Users may define their own gates using the general syntax

```
gate name(params) q, r, s {
    // operate on the quantum arguments
}
```

`q`, `r`, and `s` are quantum arguments (either qubits or registers). A user may use any number of quantum arguments greater than or equal to one in their gate definition and may give them any name. `params` represents classical

---

[1] Cross, Andrew *et al.* (2017). "Open Quantum Assembly Language."

[2] Cross, Andrew *et al.* (2021). "OpenQASM 3: A broader and deeper quantum assembly language."

parameters, which can be either numbers (for example, to define rotations) or classical bits/registers (if the gate includes a control flow). For example, here is a definition of the Hadamard gate:

```
gate h q {
    // rotate by pi/2 around y, pi around z
    U(pi/2, pi, 0) q;
}
```

`if` statements in OpenQASM 2.0 function like classically controlled gates and are written as follows:

```
/* check if both values in the
classical control register are 1 */
if (classical_register == 3) {
    h quantum_register;
}
```

My implementation of Grover's search algorithm in OpenQASM 2.0 is available here.

III. OPENQASM 3.0 SYNTAX

OpenQASM 3.0's basic syntax is very similar to that of OpenQASM 2.0, with some extensions. OpenQASM 3.0 adds several types, including `uint`, `int`, `float`, and `bool`:

```
uint unsigned_integer[8] = 42;
int signed_integer[8] = -42;
float floating_point_number[8] = 0.42;
bool boolean = true;
```

as well as the `angle` type, double-precision floating points that specifically represent angle arguments:

```
angle rotation_angle = pi/2;
```

The factor of π is not included in the actual bits that comprise the angle, but multiplied in or divided out by the compiler for ease of representation. OpenQASM 3.0 deprecates the `qreg` and `creg` keywords and replaces them with `qubit` and `bit`:

```
qubit quantum_register[n]; bit classical_register[n];
```

OpenQASM also deprecates the built-in `CX` gate, replacing it with the "gate modifiers" `inv`, `pow`, and `ctrl`:

```
// assume x is the not gate
// and is predefined
// to apply the inverse of x to q:
inv @ x q;
// to apply the square of x to q:
pow(3) & x q;
// to apply cx from r to q:
ctrl @ x r q;
```

Note that the exponent provided to `pow` must be a positive integer. OpenQASM 3.0 also adds support for while loops:

```
uint counter[4] = 0;
while (counter < 8) {
    counter++;
}
```

and for classical "subroutines," which are functions:

```
def name(params) qargs -> output {
    // body
    return output;
}
```

where `def` is a keyword, `params` are classical parameters, `qargs` are quantum arguments (their size must be specified in the subroutine declaration), and `output` is the type of the return value. For example, a function that places a four-qubit register into an equal superposition and then measures it would be implemented as follows:

```
// include the standard gate library
// which defines h, the hadamard gate
include "stdgates.inc";
def random_number_generator() qubit[4] q -> bit[4] {
    h q;
    c = measure q;
    return c;
}
```

Note that the syntax of the `measure` operator has also changed from OpenQASM 2.0, while the `include` function is the same. My implementation of Grover's search algorithm in OpenQASM 2.0 is available here.

IV. ISSUES

Both versions of OpenQASM have some readily visible issues and ambiguities. Firstly, there is no stated way for programmers to manually allocate memory; one assumes then that OpenQASM's memory allocation must be automatic, but there is no mention of garbage collection in the official documentation or white papers. Garbage collection is especially important for quantum computers because of the very small memory resources available on most machines (less than 100 qubits), so it is confusing that the developers of OpenQASM have opted for inefficient automatic memory allocation when manual memory allocation would allow programmers to better use a quantum computer's limited resources. The `reset` keyword may serve the same function as memory deallocation, since it clears a quantum bit or register, but there

is still no way to delete variables or access the hardware qubits that correspond to program variables as there is in C. OpenQASM lacks a pointer type, so the way it uses memory is opaque. This leads to the more general issue that OpenQASM's design philosophy is confusing; it claims to be an "assembly language," but its approach to memory management show that it is more high-level than even C, which its developers claim is its inspiration. Other, smaller issues include how cumbersome the gate notation is; because a quantum register is treated in the same way as a qubit is, quantum registers are not arrays of bits and cannot be treated as such by programmers attempting to write complex gates, resulting in a need for a cumbersome number of quantum arguments to be passed to a gate. This also means that programmers cannot create gates that can handle a variable number of registers; when I tried to implement Grover's algorithm, I had to lock myself into using six-qubit registers because there is no way to generalize the algorithm to n qubits because of OpenQASM's gate notation. This means that programmers who wish to scale up their algorithms must reimplement them from scratch. Overall, OpenQASM is a very powerful language that can fairly cleanly specify low-level quantum circuits, but some aspects of its design are non-ideal.